Inferring variability from customized standard software products

Nöbauer, Markus ; Seyff, Norbert ; Groher, Iris

Abstract: Systematic variability management is an important prerequisite for successful software reuse. However, it requires significant effort and extensive domain knowledge to document and maintain information on variability. In this paper we present a tool-supported approach which supports semi-automatically inferring variability information from customized standard software products. The approach does not only enable the identification and documentation of variability information based on existing products, it is also capable of incrementally updating this information. To guarantee quick access to reusable code artifacts (e.g. requirements, features or software components), the presented solution stores these artifacts together with related requirements and a generated variability model in an asset repository. The tool- supported approach has been applied to customizations of Microsoft Dynamics AX ERP systems. Our experiences highlight the potential and benefits of our approach compared to manually gathering information on software variability.

DOI: https://doi.org/10.1145/2648511.2648544

Inferring Variability from Customized Standard Software Products

Markus Nöbauer
InsideAx GmbH
4031 Linz
Austria
+43 676 897 478 400
markus.noebauer@insideax.at

Norbert Seyff
University of Zurich
8050 Zurich
Switzerland
+41 44 635 67 57
seyff@ifi.uzh.ch

Iris Groher
Johannes Kepler University
4040 Linz
Austria
+43 732 2468 4259
iris.groher@jku.at

ABSTRACT
Systematic variability management is an important prerequisite for successful software reuse. However, it requires significant effort and extensive domain knowledge to document and maintain information on variability. In this paper we present a tool-supported approach which supports semi-automatically inferring variability information from customized standard software products. The approach does not only enable the identification and documentation of variability information based on existing products, it is also capable of incrementally updating this information. To guarantee quick access to reusable code artifacts (e.g. requirements, features or software components), the presented solution stores these artifacts together with related requirements and a generated variability model in an asset repository. The tool-supported approach has been applied to customizations of Microsoft Dynamics AX ERP systems. Our experiences highlight the potential and benefits of our approach compared to manually gathering information on software variability.

Categories and Subject Descriptors
D.2.13 [**Software Engineering**]: Reusable Software – *Domain engineering, Reuse models*.

Keywords
Variability Inference, Reuse, Standard Software Product Customizations, ERP systems.

1. INTRODUCTION
Software variability management fosters systematic reuse within a family of similar products [1][2][3]. Research in the field of software product line engineering (SPLE) highlights that an adoption of SPLE typically requires dedicated processes and tool support [4]. This includes support for identifying reusable artifacts during domain engineering and creating variability models, which are utilized during application engineering to derive products as members of the product line.

SPLC '14, September 15 - 19 2014, Florence, Italy

http://dx.doi.org/10.1145/2648511.2648544

Knowledge about commonality and variability between products is often not documented explicitly. However, it is implicitly encoded in the product architecture and implementation of the various customer-specific extensions. To reduce the time and effort to create customer specific products, companies want to utilize the reuse potential of common functionality in their product portfolio.

Transitioning from ad-hoc reuse (i.e. copy and paste) to planned reuse with explicit variability management often requires changes in the development process, organization and technical realization [4][8]. Adopting a fully-fledged SPLE approach is often not feasible, especially for small companies. Even though these companies want to gain advantage from explicit variability management, they often cannot afford huge up-front investments in identifying commonalities and building reusable artifacts. Instead, they require an approach that supports a smooth transition to systematic reuse strategies supporting the reuse of existing legacy artifacts and an early payoff. The situation is even more challenging if companies customize products from other software vendors such as Enterprise Resource Planning (ERP) systems [9]. In such environments, the costs to manage variability could easily exceed the benefits. Significant platform updates typically happen every one to three years and include major architectural changes. Therefore, an established reuse infrastructure (e.g. variability models and reusable artifacts) might become obsolete every time a major update is released.

A lightweight variability management approach is required to make it possible for companies operating in such volatile environments to still benefit from variability management and planned reuse. Companies should be able to quickly identify the reuse potential in their products without the need of large up-front investments [10]. Therefore, approaches enabling a high degree of automation in creating and maintaining a reuse infrastructure are needed.

In this paper, we present an approach that enables the tool-supported inference of variability information from existing standard software product customizations. Based on the inferred variability information, an asset repository is established and maintained that contains the reusable code artifacts and related requirements. This repository is updated in an incremental and continuous way considering new products developed for customers. The results of a case study in the ERP domain highlight the potential of the presented approach. We performed an automatic variability analysis with our tool-supported approach and compared the results to a manual analysis. For identifying features,

both the manual and the automatic approach show high precision and recall. For identifying constraints between features, the automatic approach outperforms the manual analysis.

The remainder of this paper is structured as follows: In Section 2 we discuss the aim of our research and the research questions we defined. Section 3 presents our conceptual solution for inferring variability information while Section 4 presents our tool-supported approach establishing a reuse infrastructure. We also discuss limitations of our approach. In Section 5 we report on the case study we performed in the ERP domain. We revisit our research questions in Section 6. In Section 7 we discuss related work. In Section 8, we conclude the paper and give an outlook on future work.

2. RESEARCH GOALS AND QUESTIONS

The aim of our work is to develop an approach supporting the inference of variability information from existing product customizations and the establishment of a reuse infrastructure. We have specifically defined the following three research questions to guide our research:

RQ 1: To what extent is it possible to automate the inferring of variability information from existing products?

Successfully identifying variability information typically depends on the skills and domain knowledge of the analyst. Our aim is to investigate to what extend an automated approach can identify and mine variability information from existing software products.

RQ 2: What is the quality of the inferred variability information in terms of completeness and correctness?

The second research question investigates if the inferred variability information is correct compared to manual analysis of products. Furthermore, we would like to know if this information is complete.

RQ 3: What is the efficiency of the tool-supported approach compared to manual variability mining?

The focus of the third research question is to analyze whether the tool-supported approach actually provides benefits in terms of time saving.

We followed a research approach, where we first analyzed related work and the product portfolio of an SME (Small and Medium Enterprise), which is customizing standard software products in the ERP domain. As we could not identify a suitable solution to fit our needs, we developed a conceptual solution enabling the identification and maintenance of variability information based on existing products delivered to customers. We then implemented a tool-supported solution, which automatically infers variability information and establishes a repository that contains the reusable assets. In a last step we conducted a case study to provide first evidence on the validity of our approach.

3. CONCEPTUAL SOLUTION

Our conceptual solution supports inferring variability information from existing customized standard software products and incrementally building and updating a feature-based reuse infrastructure. In particular we focus on FODA-based feature descriptions [24]. Feature models in FODA support a hierarchical decomposition of features and distinguish between mandatory features, optional features and alternative features. *Requires* and *Excludes* constraints express dependencies between features. A more detailed comparison between our approach and FODA-based feature models is provided in Section 4.1.

3.1 Automatic Variability Information Inference

Figure 1 shows the artifacts and their relationships used to infer variability information in our approach. We assume that high-level requirements regarding customizations of standard software products are stored within a requirements management system (RM system) and that this information is accessible by other software tools. The requirements in the RM system contain requirements descriptions in natural language text and an associated state e.g. new, tested or delivered. Customized products contain code artifacts. A code artifact has an ID, a version number and a documentation regarding its implementation. Trace links connect requirements and code artifacts. A requirement can have zero or more trace links. Requirements without trace links are not considered in our approach. Each trace link points to one code artifact. This means there are links between each requirement and the customizations applied on a standard software product in order to implement this requirement. The trace links are bidirectional and can thus be used to identify all the requirements related to a certain code artifact.

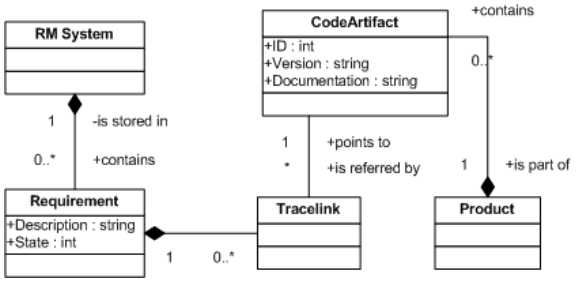

Figure 1. Artifacts used to infer variability information

Companies operating in the ERP domain typically use RM systems in the above-described way to manage customer requirements and their realizations. Commonality and variability between the different customizations is not documented explicitly but spread over the RM system. We introduce a Crawler solution, which periodically inspects the RM system and the customized product to infer this variability information:

First, the Crawler retrieves a list of requirements within a predefined scope (e.g. only those which have the state "delivered") from the RM system and sequentially inspects each requirement. The Crawler follows the trace links from the requirement to related code artifacts in the customized products. Typically, these code artifacts contain developer documentation (e.g. XML or JavaDoc comments) which is also retrieved. The requirements are typically described on an abstract level and there can be different implementations (variants of the requirements) in the products. This is done to fulfill more detailed customer specific needs. Such cases are identified as variants in our approach.

In a second step, the Crawler uses static code analysis techniques to identify additional dependencies between artifacts which have not been identified by analyzing the trace links (e.g. configuration files, other DLLs). To do so the Crawler is able to identify a set of code artifacts which are required to implement a requirement. If the Crawler has identified variants in the first step, the dependent artifacts are identified separately for each variant.

In a next step, the Crawler inspects every code artifact in the set. The trace links are used to check if one of the artifacts in the set is related to another requirement in the RM system. If such a requirement is found and all trace links from this newly identified requirement refer to code artifacts within the set, the Crawler recognizes that this requirement is also realized by the identified set of code artifacts.

Finally the Crawler combines the set of code artifacts, their requirements and all other requirements, which are implemented by the set of artifacts into one single entity. In our solution we name such an entity a feature to describe a prominent or distinctive user-visible aspect, quality, or characteristic of a standard software product [24]. The Feature Model Store (FMS) is used to store features, variability information, and relations between features in a model. The FMS fosters the reuse of this information within new projects. It provides a query interface for human domain experts to identify features and their related artifacts based on natural language requirement descriptions. The variability information within the model is used to also identify all required and recommended features. The identified features can then directly be deployed to an application prototype.

Figure 2 presents the key steps of our approach described above. Textual requirements descriptions (Requirement 1 to Requirement 4) are stored in the RM system (1). The Crawler service (2) analyses each requirement. Explicit traceability links (3) from requirements to code artifacts enable the Crawler to access corresponding code artifacts (4) within the existing product customizations (Product 1, Product 2, Product 3). Based on this information the Crawler identifies features. The Crawler inhabits a Feature Model Store (FMS) with these identified features (5). This means that the Feature Model Store contains copies of the requirements descriptions found in the RM system and copies of the code artifacts implementing them. Based on this information the Crawler infers variability information which is also stored in the Feature Model Store.

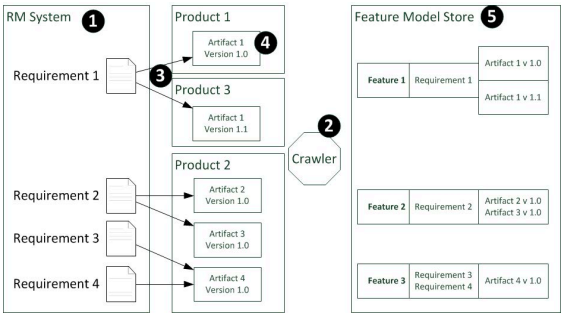

Figure 2. Conceptual Solution – A Crawler is used to populate the Feature Model Store

In Figure 2, Requirement 1 is implemented by Artifact 1 Version 1.0 in Product 1. The same requirement is implemented by Artifact 1 Version 1.1 in Product 3. This is an example of a variant. In this case our approach suggests that the Crawler combines Requirement 1 and Artifact 1 Version 1.0 and Version 1.1 as Feature 1 and makes them available in the Feature Model Store. For example, Requirement 1 describes sending electronic payment orders to a bank. While in Europe this requires SEPA format, customers in the Asia-Pacific region require a different format, such as HSBC. The two format variants are realized as different artifact versions. The high level requirement linked to the two artifact versions contains the electronic payment order description. In this case the Crawler identifies the feature "electronic payment order" with two variants, SEPA and HSBC.

In our example, Requirement 2 is implemented by Artifact 2 and Artifact 3 in Product 2. Neither Artifact 2 nor Artifact 3 are related to any other requirement. Therefore, the Crawler combines Requirement 2 and Artifact 2 and Artifact 3 to Feature 2.

Requirement 3 is implemented by Artifact 4. However, in this case Artifact 4 also implements Requirement 4. This means that the Crawler combines Requirement 3 and Requirement 4 and Artifact 4 to Feature 3 and puts it into the Feature Model Store. For example, Requirement 3 describes General Terms and Conditions to be added to the Sales Quotation letter. Requirement 4 describes additional delivery notes to be added to the Packing Slip letter. However, there exists a *Form Letter Textblock* which supports adding textual information to all customer facing documents including quotation, packing slip, invoice, etc. Both requirements can be fulfilled by the *Form Letter Textblock* feature realized in Artifact 4. Therefore the descriptions of both requirements and Artifact 4 together form a feature and are stored in the FMS.

We summarize, that in order to identify features, we follow a crawling strategy which suggests analyzing each available requirement in the RM system. In a first step, traceability links are used to identify corresponding code artifacts for the requirement under analysis. In a second step, for each of the identified code artifacts, the Crawler identifies all other requirements, which have a trace to this code artifact.

The Feature Model Store contains the identified features including the original requirement descriptions from the RM system and the reusable software artifacts. These artifacts are stored in such a way that they can be directly deployed into products. For domain analysts we provide a full-text search interface to query the requirements descriptions in the Feature Model Store. The analysts can therefore easily identify existing features in the Feature Model Store by searching for key words (e.g. SEPA). Moreover, the option to directly deploy features into a product also allows to present early prototypes to customers. We consider the Feature Model Store to be the basis for fostering systematic reuse of features.

In addition to the identification of features, the Crawler solution also supports the identification of different types of relations between features. These relations are stored within a model in the FMS. We specifically distinguish the following types of relations:

Parent-child Relationships support a hierarchical decomposition of features. Based on the information found in the RM system, we are able to identify parent-child relationships. Thereby analyzing the structure of projects and the hierarchical organization of a project's requirements as documented in the RM system in order to derive these relationships. The hierarchical structure of the features reflects the hierarchical structure of the requirements in the RM system. For example, requirements are can be related to ERP system modules such as Sales and/or Procurement. Furthermore, requirements can also describe sub modules -for example- sales requirements can be related to the sub modules Sales Orders, Payment Journals, Reports, etc.

Requires Constraints between features describe whether a feature requires another feature in order to function properly. Requires constraints can only be generated if access to the source code implementing the features can be granted. This is done via static code analysis where we investigate whether one code artifact is used by another. As these dependencies reflect the underlying

implementation technology, they are not exactly equivalent to a dependency between features from a high level perspective. However, by generating these dependencies based on artifact relations we ensure that features can be deployed from the Feature Model Store directly to other products.

Recommends Constraints describe the likelihood that two features are related, although there is no dedicated requires constraint between them (as defined above). A high percentage means that two features should be deployed together. The relation between the number of products where two features are deployed together and the total number of products under analysis are used to calculate the likelihood that these two features recommend each other. We are aware of the fact that the number of projects has a great impact on the significance of this calculation. For small numbers no useable outcome can be expected.

3.2 Incremental Maintenance

After the initial setup of the Feature Model Store, the Crawler periodically inspects the RM system and analyzes requirements and corresponding code artifacts in the available products. This is compared to the information within the Feature Model Store (i.e. the results from the last iteration). When the Crawler identifies changes, it updates the information in the Feature Model Store. Figure 3 highlights this update process.

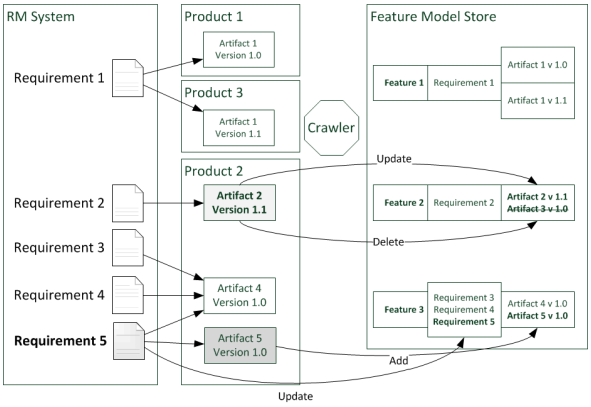

Figure 3. Conceptual Solution – A Crawler is used to update the Feature Model Store

Requirements change and products are continuously improved. After such an update, the Crawler service recognizes that Artifact 1 Version 1.0 in Product 1 and Artifact 1 Version 1.1 in Product 3 still implement Requirement 1 and that this information is already available in the Feature Model Store. No changes are required.

However, Artifact 2 in Product 2 has been updated to a higher version number (1.1). Artifact 2 in Version 1.0 is no longer in use to implement Requirement 2. A reason for such a replacement could be a bug fix in Artifact 2. So the Crawler replaces Artifact 2 Version 1.0 in Feature 2 with the current Version 1.1. Moreover, Artifact 3, which was also related to Requirement 2 disappeared. Therefore the Crawler deletes Artifact 3 from Feature 2. This could, for example, be the case if the functionality in Artifact 3 was merged with Artifact 2.

The Crawler identifies a new Requirement 5 which is also related to Artifact 4. Furthermore, Requirement 5 is linked to an additional Artifact 5. Therefore the Crawler adds Requirement 5 and Artifact 5 to Feature 3. For example, Requirement 5 describes adding Payment Instructions on the Invoice letter which is already implemented in Artifact 4 *Form Letter Textblocks*. Moreover, Requirement 5 demands saving the Invoice in PDF format and signing it electronically; this is implemented in Artifact 5. Finally, Feature 5 supports adding text to customer facing documents and signing them electronically.

3.3 Manual Refinement

We foresee that, if needed, a human domain expert can refine the information gathered automatically in the Feature Model Store. This includes providing meaningful names and descriptions for the identified features and their variants. As described in Section 3.1, variants are identified by different implementations of the same requirement in different products. The name for the identified variant as proposed by the Crawler is the name of the artifact. Although a domain expert may be able to understand the difference of certain implementations, it is certainly useful to provide more meaningful names. For example Artifact 1 Version 1.0 in Product 1 may be a class called *VendOutPaym_SEPA* while Artifact 1 Version 1.1 in Product 3 is called *VendOutPaym_HSBC*. A domain expert could provide meaningful names like SEPA Format Payment and HSBC Format Payment.

It might even be more important to refine the type of the features identified. In our approach all features are optional by default. The human domain expert may review the generated recommends constraints and their likelihood values in order to identify requires constraints. Furthermore, the human domain expert can define alternative features and exclude constraints between features. Finally, the human domain expert can also consider the companies' sales strategies e.g. group features using requires constraints and define features as mandatory. For example, the SEPA Payment feature will be a mandatory feature in all future products because electronic banking within Europe relies on SEPA. Decisions made by the human domain expert will be preserved in future updates of the Feature Model Store performed by the Crawler. In case of a conflict (e.g. domain analyst added an excludes constraint between features and the Crawler identifies a requires constraint) a notification is sent to the domain expert. Currently, in our approach conflicts need to be resolved manually.

4. CRAWLER SERVICE AND FEATURE MODEL STORE REALIZATION

We have developed a tool solution based on the concepts presented in the previous section. The overall solution is depicted in Figure 4.

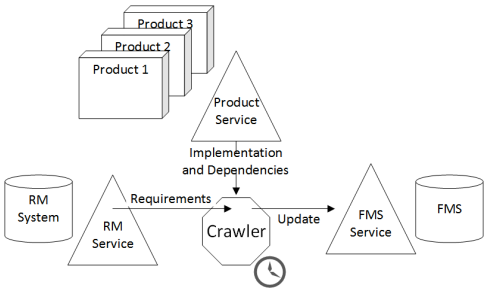

Figure 4. Technical Solution

The Crawler service is the core component of this solution and uses other services to access and store information. This includes the RM service which provides access to a given RM system, the Product service which provides access to existing products and the FMS service which enables the Crawler service to permanently store the analysis results in a Feature Model Store. All services

are implemented as XML/SOAP web services to abstract from a certain technology. This also means that the RM service and the Product service can be adapted to a given environment (e.g. a different RM system) while the rest of the solution can be used without changes. The following paragraphs describe the different components in more details.

Requirements Management (RM) Service: The RM Service provides an interface to products and their requirements. The RM service returns a list of products selected for analysis. A requirement has a unique ID, a title, a textual description and references to code artifacts implementing the requirement. Furthermore, the RM service only lists requirements that have been implemented, tested and delivered to customers. This ensures that only existing robust features will be identified in further steps of analysis.

Currently, the RM Service is able to access the Microsoft Dynamics AX[1] project module which is used to organize ERP projects and maintain associated tasks. Figure 5 shows a typical requirement. The "Effort" group contains time estimations for development and testing. The "Responsible" group is used to assign the requirements to employees. The "Task" group contains natural language text descriptions. Finally the PL4X Group is used to store a FeatureID which is used in the Version Control System (VCS) to tag modifications.

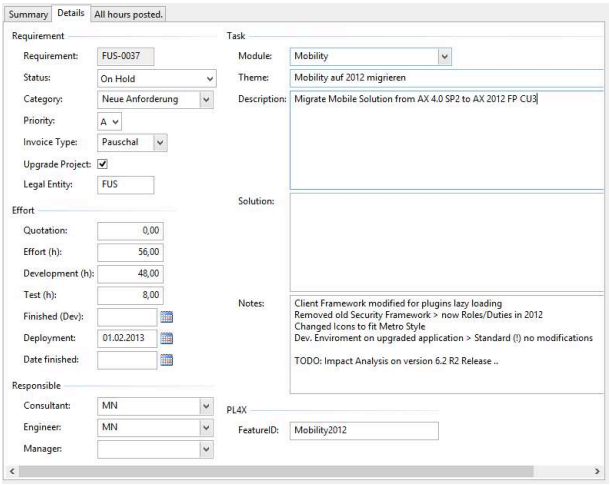

Figure 5. Requirement documented in the RM System

Product Service: The product service provides access to the customer-specific products and their code artifacts (e.g. components, table definitions, classes, macros). Furthermore it provides access to the code artifacts metadata like its version number. The product service accesses the source code and provides a static code analysis mechanism to determine dependencies between code artifacts.

In this implementation we are focusing on customized Microsoft Dynamics AX ERP products. The product service can access the source code and built-in version control system directly in the ERP system. The FeatureID from the RM system is used to identify all code artifacts, which belong to a certain requirement. Further it utilizes the ERP systems' internal development environment to resolve dependencies and identify the artifacts implementing a certain requirement.

Feature Model Store (FMS) Service and Feature Model Store: The Feature Model Store is implemented as Microsoft SQL Server database application. The FMS service is used to manipulate the model and artifacts in the Feature Model Store. It contains features, their variations and dependencies between these variations. We also support the export of variability information for import into the pure::variants[2] variant management tool. Figure 6 shows the tool prototype for maintaining the Feature Model Store. The variability information is organized as a tree with a single root. Underneath, there are mandatory features (e.g. *SEPA*), optional features (e.g. *Textblock*) and feature groups. We foresee three types of groups: The Or-Group is used to define a set of optional features. In Figure 6 the groups *Mobility*, *DMS* (Document Management Service) and *Misc* contain optional subfeatures. The And-Group is used to group a set of mandatory features. In Figure 6 the *Windream* group contains 3 mandatory subfeatures, *Client Interface*, *Server Processing* and an Or-Group (*Process*) to support typical documents. Alternative-Groups are used to define sets of mutually exclusive features. One feature can be selected precisely from the group. In Figure 6, the group *SharePoint* is such an Alternative-Group; this means either version 2010 or version 2013 is supported. Moreover, each feature can have Recommends, Excludes or Requires relations to other features in the model. In Figure 6 the selected feature *Counting* has a Requires relation to the *Item Information* feature because this feature implements logic to display an item's name, inventory on hand etc. which is used within the *Counting* feature.

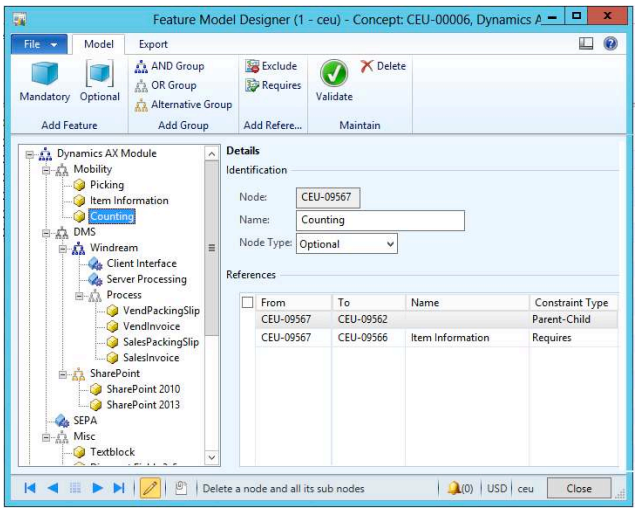

Figure 6. Feature Tree within the Feature Model Store

Crawler service: The Crawler service is implemented as timed windows service and triggers periodically to infer variability information and update the Feature Model Store. First, the Crawler service calls the RM Service to retrieve relevant requirements. In a second step, the Crawler service calls the Product Service to access the related code artifacts within the customized products. In a next step, the Crawler service updates the variability information using the FMS Service (as described in Section 3.1, Fig. 3). The Crawler service requests all the calculated require dependencies and updates the relations in the Feature Model Store via the FMS service. In a last step, the Crawler service itself calculates the recommends relations based on the code artifacts found in the products and updates the variability information via the FMS Service.

[1]http://www.microsoft.com/en-us/dynamics/erp-ax-overview.aspx

[2] http://www.pure-systems.com/Variant_Management.49.0.html

4.1 Limitations

The quality of the inferred variability information depends on the information sources, namely requirement descriptions, trace links, and code artifacts. The level at which this information is provided has a significant influence on the identification of features. For example, coarse-grained requirements might lead to coarse grained features. Also, coarse-grained code artifacts might lead to coarse grained features implementing multiple requirements. The implementation quality of code artifacts can further have an influence on the identification of constraints. For example, unnecessary and thus unwanted dependencies between code artifacts can cause the generation of requires constraints, although on the conceptual level no dependence between those features is required.

Another significant influence on the identification of features has the applied crawling strategy. We follow a bottom-up strategy to infer variability information from existing product customizations. We decided on the current crawling strategy based on our experience in the ERP domain. However, in other domains it might be needed to adjust this strategy (for example by inferring variability information from documents and other information sources).

The Feature Model Store we present supports a hierarchical decomposition of features, different feature types (mandatory, optional, alternative) and constraints between features (requires, excludes, recommends). We support the key concepts of FODA-based feature models. However, not all their concepts can be identified automatically in our approach. For example, in our tool-supported approach, the type of the identified features cannot be detected automatically. Therefore, per default, all identified features are optional. It is not possible to definitely identify features as mandatory or alternative. If a feature is included in all products under comparison, it does not mean that this feature is mandatory (even though it may be an indicator). This is similar to the identification of alternative features. If two features are never part of the same product, this does not mean that they are alternatives. Mandatory and alternative feature types can only be identified during a manual validation by a domain expert. This means that in order to provide fully-fledged FODA models, a human domain expert is needed to finalize the inferred information and export it to pure::variants for a convenient graphical representation.

While requires constraints can be automatically identified based on artifact dependencies, our approach cannot automatically identify excludes relationships. If two features are never part of the same product, this does not mean that they are mutually exclusive. Such relationships can again only be identified by a domain expert. In addition to FODA we introduce recommends constraints. The calculated likelihood for the presence of a feature can support the domain expert in selecting relevant features for a new product instance.

5. EVALUATION AT INSIDEAX

The tool-supported approach of inferring variability information described in this paper has been initially evaluated in cooperation with InsideAx, an Austrian SME. Using variability information for strengthening systematic reuse is a key aim of InsideAx as they customize ERP solutions from a large software vendor (Microsoft). The vendor frequently provides updates including major system changes. Therefore, the costs of creating and maintaining a reuse infrastructure could easily exceed the benefits. The automation approach presented is seen as a solution to this problem which could outweigh the time and effort needed for manually identifying and maintaining variability information.

5.1 Evaluation Method

The goal of the evaluation was an initial proof-of-concept demonstration regarding the automatic inference of variability information as supported by our approach. Furthermore, we aimed at getting first feedback on the utility of the tool-supported approach. We conducted a study at InsideAx focusing on comparing variability information provided by a domain expert to variability information that was automatically inferred by our Crawler. We compared, in particular, the quality of the information inferred in terms of correctness and completeness (RQ2) and investigated the efficiency of the automated approach (RQ3) by analyzing the time needed for variability identification.

For a first evaluation we focused on one customized product and compared it to the standard product. We followed a comparison strategy where we involved one consultant at InsideAx in order to manually create a feature model for customizations developed for Dynamics AX 2012 ERP. The bespoke consultant has a master's degree in computer science and more than two years' experience in ERP system customization. He is neither an author of this paper nor was he aware of the automated approach discussed in this paper. The consultant was given the task to build a FODA based variability model using the pure::variants feature modeling tool. He was familiar with the ERP system and pure::variants and had access to all artifacts relevant for his task (e.g. requirements and code artifacts). The consultant himself tracked the time needed for creating the variability model.

In parallel, one of the authors of this paper, who is also the developer of the Crawler service, used the automatic approach to infer variability information for the same system and its customizations. We tracked the time needed for automatically inferring that information. To make the results of the automated approach comparable to the manual analysis the variability information gathered and stored in the FMS was exported to a pure::variants file format. Like the other consultant, the author is a domain expert in this field and has 8 years of experience in customizing ERP systems.

After the creation of the two feature models, a manual model analysis and comparison has been conducted in terms of model correctness and completeness as discussed below. This was done by the two model creators and another domain expert from InsideAx. The results were reviewed and checked by the other authors of this paper.

$$Recall = \frac{|\,Identified\ Elements \cap Relevant\ Elements|}{|\,Relevant\ Elements\,|}$$

$$Precision = \frac{|\,Identified\ Elements \cap Relevant\ Elements\,|}{|\,Identified\ Elements\,|}$$

Figure 7. Metrics for evaluating correctness and completeness

As there was no approved feature model available for the system under analysis (to serve as ground truth), the workshop participants compared the manually and automatically generated models and discussed the deltas between them. As pure::variants as well as the Feature Model store support similar concepts such as features, requires- and recommends dependencies, both models could be compared without information loss. In a first step we focused on the features and compared their number and overlap within both models in order to check if the same features had been identified. We would like to stress that we did not compare the type of the features identified, as in this particular system features are always optional per default. In a second step we compared the hierarchical model structure by reviewing the correctness of parent-child relationships for each model. The third step focused on reviewing the relations identified (i.e. requires constraints) in both

models separately in order to validate their correctness. To this end, we performed a manual code analysis.

This analysis also served as a basis for the calculation of correctness and completeness using metrics such as precision and recall (see Figure 7). Furthermore, the workshop was used to also discuss the applicability of our approach at InsideAx.

5.2 Results

The results of the syntactical analysis revealed that both models were correct. The automatically generated model was successfully exported to pure::variants where its syntactical correctness was confirmed by the domain experts. In the case of the manually created variability model, the syntactical correctness was ensured by using pure::variants as modeling tool and confirmed by the domain experts.

The results of the more detailed feature evaluation revealed that the automatically generated model consisted of 82 features. However, there were 8 false positives (features identified by the Crawler service which were not actually features). These false positives had already been identified within the initial model check after creation; no comparison with the manually created model was needed. Furthermore, 5 valid features, which were identified using the manual approach, were not identified by the Crawler. In total, the manually created model consisted of 66 features. There were no false positives but comparing this model to the automatically generated one we found that 13 valid features were missing. We will discuss the reasons for these differences between the manual and the automatic identification of features in the next section.

A detailed analysis of the hierarchical model structure revealed that the parent-child relationships identified were correct in both models. In order to come to this conclusion we had to review the relevant information artifacts in the RM system.

The analysis of the relationships within the models showed that the tool-supported approach identified 64 requires dependencies between features with the help of static code analysis. All of them were considered to be valid dependencies. Manual variability modeling only led to the identification of 6 requires dependencies. As we did not compare multiple customized products, the Crawler did not generate recommends relations. However, the consultant identified one recommends dependency based on the textual requirements description.

Table 1. Evaluation Results

	Features					Constraints				Time needed for model creation/generation
	# identified	# false positives	# missing	Recall	Precision	# require	# recommend	Recall (including require and recommend dependencies)	Precision (including require and recommend dependencies)	
Manual	66	0	13	84%	100%	6	1	11%	100%	4 hours
Generated	82	8	5	94%	90%	64	0	98%	100%	30 min.

From these results, we were able to calculate Recall and Precision regarding the identification of feature and relations (see Table 1). Our calculations show that the manually created model has a high correctness ratio. Every feature or relationship identified was correct. However, the manually created model is less complete than the automatically generated model. The Crawler service was able to identify more valid dependencies by utilizing the product service' static code analysis capabilities, which lead to a high completeness value. The completeness value for the identified features is also higher when compared to the manual approach.

Both the consultant and the Crawler service used the RM system as initial input. The consultant needed 4 hours to come up with a variability model (see Table 1). In a first step the consultant spent about 30 minutes to identify features but he was able to skip incorrect data from the RM system. The consultant spent more than 3 hours to incrementally build the variability model by identifying features and their dependencies which leads to the 4 hours spent in total. The author, who used our tool-supported approach, needed 5 minutes for initial setup activities (e.g. configuring the interfaces to RM and Product Service). The actual inferring of the variability information needed less than 25 minutes. This included fetching requirements from the RM system, following trace links and identifying dependencies, generating features and placing them in the Feature Model Store.

5.3 Findings

The results reveal that our solution enables the automatic inference of variability information at a high quality compared to a manual variability analysis. However, false positives were identified which could be traced back to corrupt data in the RM system. For example, by mistake an internal developer meeting was scheduled as requirement instead of appointment. This meeting belonged to a customized product and had a reference to code artifacts which were debugged during this meeting. Therefore, the RM services identified it as relevant requirement, the product service found its referenced code artifacts and the Crawler service identified it as new feature. Furthermore, the automated solution missed 5 features found in the manually created model. A more detailed analysis revealed that, again, incorrect status descriptions in the RM system caused these problems. The requirements state of bespoke features was not properly set (e.g. state "in concept" instead of "deployed"). While the consultant recognized the actual state (e.g. deployed) and included these features in the variability model, the Crawler service skipped them.

A more detailed discussion between the InsideAx consultants also suggests why manual analysis had to cope with problems. In cases where the requirement was poorly documented in the RM system (e.g. missing textual description), the consultant excluded it from his analysis resulting in missing features in the variability model. Furthermore, the consultant did not manually perform an in depth code analysis due to time issues and therefore failed to identify several valid constraints.

Within the analysis of the results the consultants at InsideAx also revisited the source data in the RM system and corrected the requirements state and type. This means that in a next iteration the Crawler service will delete the false positive features identified and will add the missing features to the automatically generated variability model.

The consultants at InsideAx also came to the conclusion that, following our approach, the quality of the model generated depends on the quality and granularity of requirements documented in the company's RM systems. Periodical checks of automatically generated variability models are needed. However, as the Crawler service uses data from the RM system to maintain the variability model, it can also be used to monitor the quality of the information within the RM system. By correcting problems in the RM system, the Crawler also corrected automatically the variability model.

As for the initial evaluation, consultants at InsideAx claimed that a tool such as the Crawler service could support their daily work

and was seen as a useful alternative to manually creating variability models. They stressed that the approach supports the automatic and incremental generation of a variability model from the very first customized product and maintains the variability model as the number of customized products increases. Furthermore, they stated that the information provided in the Feature Model Store would support reuse of existing artifacts in future projects. However, we need mechanisms to search for relevant features in the Feature Model Store and to deploy selected features to new products.

5.4 Threats to Validity

The validity of the results reported was subject to the following possible threats. Threats regarding conclusion validity (i.e. issues that affect the ability to draw conclusions) include the limited number of experiments and the limited number of products analyzed. The automated approach was applied for one particular example where we compared an automatically generated model with a manually created one for a single product. We did not monitor or compare the model generation on a long-term scope for multiple customized products. Furthermore, results were compared with only one manually created variability model. This of course does not allow drawing any statistically relevant conclusions. However, initial results are promising and serve as a first proof-of-concept regarding the automatic generation of variability models with the help of our tool-supported approach.

Internal validity of the study (i.e. identifying factors which actually caused the observed effects) is challenged by the fact that one author of this paper, who is familiar with the developed tool-supported solution, also applied it in the case study. To overcome this, the time required to setup the Crawler service and its components was not included in our calculations. People who are not familiar with the tool-supported approach might need some time to understand and apply it. However, we also did not include the time needed for learning how to use the pure::variants feature modeling tool in order to be able to create a model. We consider these to be one-time investments in a startup phase and therefore not critical, also assuming a long-term application of a particular approach.

We consider the discussion of threats regarding external validity (i.e. issues that limit the generalization of the results) of high importance. The first evaluation was conducted in a particular setting matching the capabilities of the approach developed. We expect the results presented can be generalized for similar environments. We consider this setting to be common in domains where standard products are customized for particular customers (e.g. in field of ERP) but we are also aware that there exist other settings where our solution cannot be applied as is. In those cases our solution would have to be adapted to reflect the particular setting.

6. RESEARCH QUESTIONS REVISITED

We would like to present answers to our research questions in the following section.

RQ 1: To what extent is it possible to automate the inferring of variability from existing products?

Regarding RQ1 we have shown that the tool-supported approach presented in this paper allows the automatic inference of variability information from existing customized standard software products. However, there are several limitations and constraints. This approach requires information about existing products to be available. For example, explicit trace links between requirements and code artifacts need to be provided. Furthermore, the identification of parent features is based on information available in the RM system. She et al. [13] discuss other options to identify parent features but also highlight the complexity of this task. We conclude that our approach works for the setting and environment described. By fostering an adaptable and extendable solution, we are confident that further extensions will also allow its application in other contexts.

RQ 2: What is the quality of the inferred variability information in terms of completeness and correctness?

The evaluation we conducted provides first answers to RQ 2. The results of the first case study highlight that the developed tool - supported approach is able to generate variability information of high quality in terms of completeness and correctness compared to the manual approach. However, we still need a final check of the automatically generated information by a human analyst. The results of this initial evaluation highlight that most issues regarding inferring variability information could be traced back to misleading information in the RM system. However, the evaluation we conducted did not allow us to fully test our automated approach. Due to the limited number of systems under analysis, we could not test the calculation of recommend dependencies. The iterative update of the variability information was not covered. Further evaluation is needed to provide statistically relevant answers.

RQ 3: What is the efficiency of the tool-supported approach compared to manual variability mining?

Regarding RQ 3, we can conclude that the effort to infer variability information from existing customized products is significantly lower using the tool-supported approach. In comparison to a manual approach we see great potential in terms of time savings. Although the case study we conducted was only a small example for customizations in the ERP domain, the human analyst needed 4 hours to create the feature model. In contrast, the initial tool setup required approximately 5 minutes; the tool calculated about 25 minutes where no human interaction was needed and the manual check took 10 minutes. We are aware that the time needed to refine the generated variability information varies. In other domains with more mandatory and alternative features in place, a human domain expert would need more time to refine the model than in the ERP domain with many optional features. However, we expect that a human domain expert will benefit from the generated recommends constraints and therefore will require significantly less time refining the information than creating one manually.

Regarding scalability of our approach, we are aware that the time needed to automatically infer variability information will increase with the number of products and requirements. This also means that the time needed to refine the generated information will increase. However, we do not see this as an issue as we expect the time for refinement to be significantly lower than manual variability modeling.

7. RELATED WORK

Closely related to our approach are the areas of *(automatic) variability analysis*, *variability mining* and *feature identification*:

Xue [17] proposes an automated variability analysis by combining both feature knowledge from a top-down domain analysis and a bottom-up analysis of clones in software products. The top-down analysis requires a set of product feature models as input that captures the features contained in a product including their dependencies. The comparison of the models is based on lexical and structural similarities of features. Our approach creates a variabil-

ity model based on requirements stored in a RM system and links to product artifacts. No further input in form of product feature models is required. Our analysis does not explicitly take into account similarities on implementation level.

Ziadi et al. [18] present a three-step approach for the identification of features from different software products. In the first step, a higher level representation (similar to a UML class diagram) of the source code of each product is generated. In the second step, feature candidates are computed based on a similarity analysis of the models obtained in the first step. In a final step, a manual analysis is performed that identifies non relevant and missing features. In our approach, the higher level of requirements is the basis for the variability analysis. We analyze commonalities between the requirements in the different products and generate a variability model from the results. No analysis of the source code is performed in our approach because the similarity between the artifacts is implicitly encoded in the version number. Also, we do not explicitly include a manual analysis step but it would easily be possible to add or remove variability in the Feature Model Store.

Alves et al. [20] propose a framework for the development of feature models using information retrieval techniques for identifying variability in requirements specifications. First, a set of requirements documents that each contain a requirements specification of an application are analyzed for similarity and combined into clusters. Next, configurations (instantiated feature models) are abstracted from the result of the previous step. A feature model is created by merging the different configurations. Weston et al. [19] also perform an analysis of natural-language requirements documents to obtain a feature model. The documents are merged to produce a hierarchy of features which are clusters of requirements. This text is then mined for potential variability elements within the document. The engineer then has to decide how to represent this variability in the feature model. John [23] presents PuLSE-CaVE, an approach for the extraction of requirements from existing user documentation. Common and variable requirements fragments are extracted from the documents based on extraction patterns the engineer selected. In a last step, the requirements that have been extracted are validated by a domain expert. In our approach, the requirements stored in the RM system are the primary source of information. As in [19] and [20] we define features as sets of requirements. The feature model is, in our case, not obtained by performing a similarity analysis of different application requirements but by analyzing the assets and their versions related to the different requirements.

The FLAT^3 tool [17] supports feature location by applying both information retrieval techniques and execution trace collection. Feature location involves textually searching a project's source for code that is similar to a query that describes a feature. Dynamic feature location entails running the software and invoking the feature of interest to capture a trace of the source code that was executed. Our approach is not based on textual similarity analysis but on trace links between requirements descriptions and code artifacts. Results of the feature identification are stored. Our approach stores the feature and its code artifacts together in what we call the Feature Model Store.

Acher et al. [21] present an approach for the generation of feature models from product descriptions in tabular format. The input to the process is a table comparing the different products and user input on how the data should be interpreted. Feature models of the different products are generated based on this information and they are merged into a feature model of the set of products. The generation of the variability model is in our approach based on the requirements and their links to different versions of code artifacts. The requirements together with the links represent the product information. As opposed to our approach, in [21] product artifacts apart from the product descriptions are not taken into account.

She et al. [16] present an approach for reverse engineering feature models. The tool is capable of assisting domain experts in constructing the feature hierarchy and automatically identifying feature groups and its' requires and exclude relationships. A list of feature names and their descriptions is required as input to the process. Those are extracted from existing documentation. Also, a formula describing feature dependencies needs to be provided. Our approach only requires multiple product implementations and their requirements descriptions. No further information about feature dependencies is required. As opposed to [16], our approach is not capable of identifying feature groups and requires and excludes relationships between features.

Acher et al. [22] also present an approach for reverse engineering feature models. First, a feature model of the architecture that represents a hierarchy of components is extracted from existing code artifacts. The model contains all components available in the system even if they potentially exclude each other. Based on an analysis of the specifications of the components and their dependencies, an additional feature model is created that also contains dependencies (requires, excludes). These two models are then aggregated to obtain an architectural feature model that represents the architectural variability. Our approach analyzes the requirements and implementation. Dependencies between features are obtained from their occurrence in the different products.

The LEADT approach [15] supports locating, documenting and extracting implementations of product-line features from legacy code. Developers need to identify initial seeds for the features in the legacy code base. The identified code is iteratively expanded until the complete feature is extracted. While the LEADT tool assists developers in extracting feature code, the features and their relationships need to be identified manually. Our approach automates the process of variability model development and also extracts the components implementing the features.

In [25] Haslinger et al. present an approach to extract a feature model from valid set of feature combinations. As in our work, they follow a bottom-up approach and build a model based on existing products. However, in contrast to their work we do not infer the structure of the model from the identified features but base it on the existing structure in the RM system.

8. CONCLUSION AND FUTURE WORK

In this paper, we present our approach to automate inferring and maintaining of variability information. We describe a solution which is based on a Crawler using existing information in an RM system and existing customized products; this, in order to generate variability information stored in a Feature Model Store together with requirements and code artifacts. Furthermore, we present a first evaluation study for our solution in the ERP domain. We compared the automatic variability inferring approach to a manual variability model creation. We analyzed the structure of the models and compared the number of features and relations. Furthermore, each model was validated against the actual system at hand.

We conclude that, being able to automatically infer variability information based on existing information about software products. Future work will focus on two directions: (i) extending the approach to make it applicable in other domains and settings. We presented a crawling strategy that generates suiting feature granularity in the ERP domain. However, this strategy has to be adjust-

ed for other domains for the quality of the information artifacts and implementation quality. We will also conduct more in depth evaluations and (ii) providing mechanisms and tools which enable practitioners to use the information available in the Feature Model Store to foster structured and systemic reuse of these artifacts.

9. REFERENCES

[1] Pohl, K., Böckle, G., and van der Linden, F. Software Product Line Engineering. Foundations, Principles, and Techniques. Springer-Verlag Berlin Heidelberg, 2005.

[2] Clements, P., and Northrop, L. M. Software Product Lines: Practices and Patterns. Addison Wesley, 2007.

[3] Van der Linden, F. J., Schmid, K., and Rommes, E. Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering, Springer, 2007.

[4] Weiss, D.M. and Lai, C.T.R. SW Product Line Eng. A Family-Based SW Dev. Process. Addison-Wesley, 1999.

[5] Hilliard, R. On Representing Variation. 1st International Workshop on Variability in Software Product Line Architectures, 4th European Conference on Software Architecture, Copenhagen, Denmark, pp.312-315, 2010.

[6] Galster, M., Avgeriou, P., Weyns, D., and Männistö, T., "Variability in Software Architecture: Current Practices and Challenges". ACM SIGSOFT SW Eng. Notes 36(5), 2011.

[7] Galster, M. and Avgeriou, P., "The Notion of Variability in Software Architecture – Results from a Preliminary Exploratory Study", 5th Int. WS on Variability Modelling of SW-Intensive Sys. (VaMoS), Namur, Belgium, pp. 59-67, 2011.

[8] Böckle, G., Bermejo, M.J., Knauber, P., Krueger, C.W., Leite, J., van der Linden, F., Northrop, L.M., Stark, M., and Weiss, D.M., Adopting and Institutionalizing a Product Line Culture, 2nd International Conference on Software Product Lines (SPLC), San Diego, USA, pp. 48-59, 2002.

[9] Nöbauer, M., Seyff, N., Dhungana, D., and Stoiber, R., Managing Variability of ERP Ecosystems: Research Issues and Solution Ideas from Microsoft Dynamics AX, 6th Int. Workshop on Variability Modelling of Software-intensive Systems (VaMoS), Leipzig, Germany, pp. 21-26, 2012.

[10] Nöbauer, M., Seyff, N., Groher, I., and Dhungana, D., A Lightweight Approach for Product Line Scoping, 38th Euromicro Conf. (SEAA), Cesme, Tukey, pp. 105-108, 2012.

[11] Kock, N., "Information Systems Action Research: An Applied View of Emerging Concepts and Methods", Integrated Series in Information Systems, vol. 13, Springer, 2007.

[12] Quigley, B.A., Kuhne, G.W., Creating Practical Knowledge through Action Research: Posing Problem, Solving Problems, and Improving Daily Practice, Jossey-Bass, 1997.

[13] Kang, K. C., Cohen, S. G., Hess, J. A., Novak, W. E., and Peterson, A. S., Feature-Oriented Domain Analysis (FODA) Feasibility Study, Technical Report, SEI, 1990.

[14] Kästner, C., Dreiling, A., Ostermann, K.. Variability Mining with LEADT, Tec. Rep., Philipps Univ. Marburg. 2011

[15] She, S., Lotufo, R., Berger, T., Wkasowski, A., and Czarnecki, K., Reverse Engineering Feature Models, 33rd International Conference on Software Engineering (ICSE), Waikiki, Honolulu, USA, pp. 461-470, 2011.

[16] Savage, T., Revelle, M., and Poshyvanyk, D., FLAT^3: Feature Location and Textual Tracing Tool, 32rd International Conference on Software Engineering (ICSE), Cape Town, South Africa, pp. 255-258, 2010.

[17] Xue, Y., Reengineering Legacy Software Products into Software Product Line Based on Automatic Variability Analysis, 33rd International Conf. on Software Engineering (ICSE), Waikiki, Honolulu, HI, USA, pp. 1114-1117, 2011.

[18] Ziadi, T., Frias, L., da Silva, M.A.A., and Ziane, M., Feature Identification from the Source Code of Product Variants, 16th European Conf. on Software Maint. and Reengineering (CSMR), Szeged, Hungary, pp. 417-422, 2012.

[19] Weston, N., Chitchyan, R., and Rashid, A., A Framework for Constructing Semantically Composable Feature Models from Natural Language Req., 13th Int. SW Product Line Conf. (SPLC), San Francisco, USA, pp. 211-220, 2009.

[20] Alves, V., Schwanninger, C., Barbosa, L., Rashid, A., Sawyer, P., Rayson, P., Pohl, C., and Rummler, A., An Exploratory Study of Information Retrieval Techniques in Domain Analysis, 12th International Software Product Line Conference (SPLC), Limerick, Ireland, pp. 67-76, 2008.

[21] Acher, M., Cleve, A., Perrouin, G., Heymans, P., Vanbeneden, C., Collet, P., and Lahire, P., On Extracting Feature Models from Product Descriptions, 6th International Workshop on Variability Modeling of Software-Intensive Systems (VaMoS), Leipzig, Germany, pp. 45-54, 2012.

[22] Acher, M., Cleve, A., Collet, P., Merle, P., Duchien, L., and Lahire, P., Reverse Engineering Architectural Feature Models, 5th European Conference on Software Architecture (ECSA), Essen, Germany, pp. 220-235, 2011.

[23] John, I., Capturing Product Line Information from Legacy User Documentation, Software Product Lines - Research Issues in Eng. and Mgmt, pp. 127-159, Springer, 2006.

[24] Kang, K., Cohen, S., Hess, J., Novak, W., and Peterson, A., "Feature-Oriented Domain Analysis (FODA) Feasibility Study," Software Eng. Institute, Carnegie Mellon Univ., Pittsburgh, US-PA, Tec. Rep. CMU/SEI-90-TR-021 , 1990.

[25] Haslinger E., Lopez-Herrejon R., Egyed A., On Extracting Feature Models from Set of Valid Feature Combinations, 16th Int. Conf. on Fundamental Approaches to Software Engineering (FASE), Rome, Italy, pp. 16-24,